Procedia Computer Science

Volume 51, 2015, Pages 1463–1472

ICCS 2015 International Conference On Computational Science

Prefetching Challenges in Distributed Memories for CMPs

Martí Torrents[1], Raúl Martínez[2], and Carlos Molina[3]

[1] Computer Architecture Department, UPC - BarcelonaTech, Barcelona, Spain
martit@ac.upc.edu

[2] Computer Architecture Department, UPC - BarcelonaTech, Barcelona, Spain
raulm@ac.upc.edu

[3] Computer Engineering Department Universitat Rovira i Virgili, Tarragona, Spain
carlos.molina@urv.net

Abstract

Prefetch engines working on distributed memory systems behave independently by analyzing the memory accesses that are addressed to the attached piece of cache. They potentially generate prefetching requests targeted at any other tile on the system that depends on the computed address. This distributed behavior involves several challenges that are not present when the cache is unified. In this paper, we identify, analyze, and quantify the effects of these challenges, thus paving the way to future research on how to implement prefetching mechanisms at all levels of this kind of system with shared distributed caches.

Keywords: Prefetching, Networks on chip, Distributed memories, CMPs, Challenges

1 Introduction

An imbalance in technological improvements in recent years has led to an increasing gap between processor and memory speeds. One way to address this problem is to use latency hiding techniques such as prefetching [9]. This mechanism tries to predict which data the processor will require in the near future and bring it to the nearest cache level before it is needed by the application. Prefetching is a key technique employed by almost all current commercial high-performance processors in at least one but typically all levels of their memory hierarchy [1]. Recently, processor design techniques have evolved toward architectures that implement multiple processing cores on a single die (CMPs). In this way, future CMPs with tens (or even hundreds) of processor cores will probably be designed as arrays of replicated tiles connected over an on-chip switched direct network [11]. The most common memory hierarchy organization of these architectures lies on one or two private cache levels per tile and, as a last level of cache, a Distributed and Shared Memory (DSM). This DSM holds a banked organization with one bank on each tile. Moreover, each cache block-sized unit of memory is statically mapped to one of the banks based on its address in an interleaved way. In these systems, if prefetching is allocated on a DSM cache level, the prefetcher also has to be distributed in such a way that each

Selection and peer-review under responsibility of the Scientific Programme Committee of ICCS 2015

doi:10.1016/j.procs.2015.05.336

tile of the CMP contains a prefetch engine. In a straightforward implementation derived from unified memory systems, these prefetch engines would work independently from each other by analyzing the set of memory accesses that arrive at the attached piece of cache and potentially generate prefetching requests targeted to any other tile on the system depending on the target address. We have observed that this distributed behavior entails several challenges that are not present when the cache is unified. In order to better understand these challenges, we introduce the following concepts:

- Memory access pattern detection: The vast majority of hardware prefetching mechanisms are based on structures that store the recent memory-related activity of the processor. Using this information and dedicated hardware, the prefetcher predicts which data the processor is going to request and takes these data to the nearest cache level before they are requested by the application. There have been a lot of different prefetching engine variants proposed in the literature for single core [12] and multi-core environments [9].

- Prefetching effectiveness parameters: The effectiveness of prefetching depends greatly on the correct prediction of future memory accesses of the specific mechanisms used but also on other parameters like the timeliness of the generated requests. Note that prefetching a request may contaminate the cache by substituting useful memory lines with other lines that may or may not be needed in the future. There are several metrics that are used to measure prefetching effectiveness [10]. The most important of these parameters is the accuarcy, which is calculated by dividing the *useful prefetchers*[1] by the *total number of prefetchers*[2]. This metric is usually tracked by adding an extra bit per cache line that is set for prefetched blocks. If one of those lines is used by a demand operation, a useful prefetch is counted.

- Dynamic management: Given that the effectiveness of prefetching depends on a high degree on workload and run-time effects, several techniques have been proposed to dynamically adapt the behavior of the prefetching mechanism based on run-time profiling information of the previously presented parameters. The most relevant techniques that have been proposed are **filtering** [14], [13], **throttling** [2], and **prioritization** [5], [7]. Nevertheless, prefetching engines employed in these works, and as far as we know in any other area of the literature, are attached to unified memory levels (typically L1) and in no case to different pieces of a distributed one.

The challenges introduced by the distributed behavior are: (1) **pattern detection**, (2) **prefetching queue filtering**, and (3) **dynamic profiling**. In this paper, we identify, analyze, and quantify the effects of these three challenges. Moreover, we provide a comprehensive study which help future research to solve these issues. However, presenting a solution is out of the scope of the paper. As we will show in the following sections, if these challenges are not properly addressed, the efficiency of the prefetching will be reduced due to non-detected or wrongly detected patterns, and redundant requests traveling through the network. Furthermore, dynamic management techniques that rely upon effectiveness parameters measurement will be less effective due to inaccuracies in the profiling of these parameters. The objective of this work is to provide a light in the research directions that researchers should follow.

[1] Useful prefetches are the number of prefetching operations that bring a cache line into the cache and this cache line is subsequently used by a demand operation.

[2] Total number of prefetches are the total number of memory operations launched by the prefetch module.

The paper is organized as follows. Section 2 presents the challenges studied in this paper. Section 3 shows the modifications performed in the simulator to evaluate the challenges. Section 4 describes the evaluation framework and quantifies the impact of these challenges. Section 5 provides some research directions for solutions to the challenges. Finally, Section 6 summarizes our main conclusions.

2 Challenge identification

To identify the challenges, we divide the work done by a prefetcher into three phases: (1) analysis, (2) request generation, and (3) evaluation. Figure 1 shows four tiles, which represent the behavior of the prefetcher in each phase. Note that we have tagged the arrows with a number that represents the behavior in each phase.

1. **Analysis phase**: The prefetcher collects the information related to the memory accesses and analyzes it. To do this, some prefetchers use internal structures that are filled with information related to the memory accesses. The analysis then determines whether prefetch requests must be generated. If prefetch requests are generated, the analysis phase is responsible for deciding the number of generated requests and the memory blocks that must be prefetched.

2. **Request generation phase**: At the end of the analysis phase, several requests may be generated. In this phase, the requests are queued into the prefetch queue. Before being queued, the prefetch queue checks whether the requests are already in it. If they are, they are merged and not queued. Once in the queue, the requests will wait until the cache controller pops them from the queue to issue the requests to the memory hierarchy.

3. **Evaluation phase**: This phase takes place when a request is completed. At that moment statistics are taken dynamically for each request. This profiling information is used in the analysis phase by some dynamic techniques to modify the variables that the prefetcher is working with.

In the following, we explain the challenges we have detected. Notice that each challenge directly affects one of the previously described phases of the prefetcher. We can therefore say that our study covers all the phases of any prefetching mechanism.

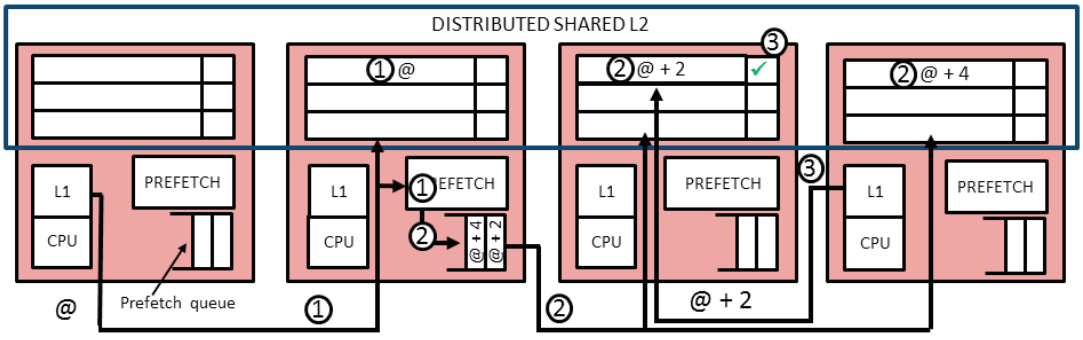


Figure 1: Phases from the prefetcher in a distributed memory system: (1)analysis, (2) request generation, and (3) evaluation.

2.1 Pattern detection

This challenge appears in the analysis phase. Many prefetchers usually save the history of memory accesses in internal structures in order to analyze the stream of accesses and, with this information, generate the prefetching requests. The problem is that the prefetcher is distributed in the same way as the memory. This means that each prefetcher can only be aware of a certain part of the stream of accesses. Figure 2 shows an example of how a stream may be distributed. Notice that prefetchers that work in unified memory are able to analyze the full memory pattern. However, as we can see, in DSM each prefetcher can only be aware of a certain part of the pattern. This distribution does not allow the prefetcher to make accurate analysis, so the prefetcher may not work properly (it may launch fewer requests and/or be more inaccurate).

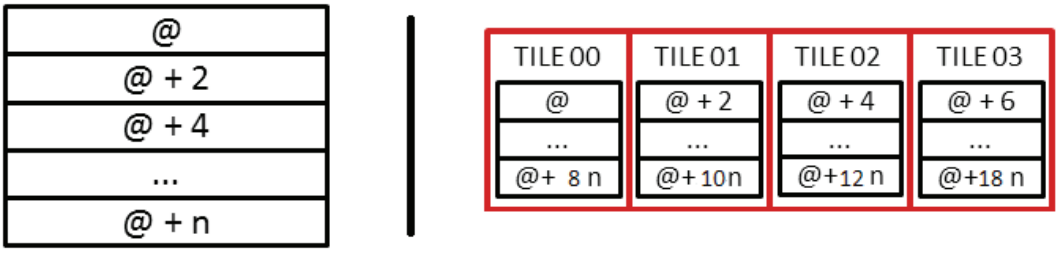


Figure 2: A possible distribution of a memory accesses stream (@, @+2, @+4, ..., @+n): on the left in a unified memory and on the right in a 4-tiled distributed memory.

2.2 Prefetching queue filtering

The prefetching queue filtering challenge appears in the request generation phase. In a unified memory system, there would be only one prefetch queue in which all the prefetch requests can be queued. The merging operation implemented on this queue may filter many recently repeated prefetch operations. However, if these operations are not filtered, they will be sent to the tile where they are addressed, which will waste power and increase the network contention. Moreover, once they are in the tile they will most probably realize that the data is already in the memory. In the DSM, instead of only one prefetching queue, there is a prefetching queue in each tile. This means that repeated prefetch requests may exist in several prefetching queues. However, as there is no communication between the prefetching queues, these requests will not be filtered.

2.3 Dynamic profiling

This challenge appears in the evaluation phase of the prefetcher. When the prefetcher attached to a tile starts generating requests, these requests may be addressed to any other tile in the system depending on the address. This means that the statistics related to a prefetch request are not collected in the same tile where this prefetch request is generated. For this reason, the statistics that a tile is recording are not related to the prefetcher allocated in this tile. These statistics are a collection from the prefetch requests received by this tile from the prefetchers in all of the tiles. This means that if the prefetcher or some management technique needs to use these dynamic statistics and collects the statistics in its tile, they will most probably be inaccurate and the decisions taken based on them may be wrong.

3 Challenge evaluation methodology

To evaluate the impact of the challenges identified in the previous section, we needed to modify several parts of the simulator. In the following subsections, we explain how we managed to quantify each challenge.

3.1 Infrastructure for pattern detection

To quantify this challenge, we developed an ideal prefetcher. This ideal prefetcher is centralized, gathering information from all the caches in a single set of shared structures and trying to recover the memory correlation from each processor. This prefetching system monitors the memory accesses in any cache module at the time they are generated and without generating any congestion in the network. To guarantee this, each tile has a direct connection with the prefetcher. Note that, the correlation and the patterns that the prefetchers try to predict are in the order that the core is executing the memory instructions. However, the DSM is receiving requests from all the cores at the same time without any order. For this reason, the centralized prefetcher would not be able to recover the correlation from the memory access stream. To solve this problem, the ideal prefetcher classifies each memory request, according to its execution core. There is one prefetcher per each core, thus when a memory operation is detected, it is assigned to the prefetcher related to its execution core. When a prefetch request is issued, it is queued in the prefetch queue of the tile where it has to be resolved, so no extra operations are needed. When a cache controller is able to issue a prefetch operation, it will check if there is any operation to issue in its prefetching queue. If so, the prefetch operation will be issued. Note that, as the prefetcher is ideal, the costs in terms of time, power, and the overhead of all these connections and switches are not taken into account.

3.2 Infrastructure for queue filtering

To quantify this challenge, we implemented in the simulator a global buffer that holds the list of pending requests in each of the prefetching queues. With this centralized information, when queuing a request, the prefetch queue can detect whether this request is in any other tile and merge it. Obviously, this buffer is used only to quantify this challenge. For this reason, the communication between the buffer and the tiles is done instantaneously and without congesting the network. The requests generated by the prefetchers in the cores are sent to the global prefetching queue. The mechanism to issue the prefetch operations is the same as in the previous implementation. To quantify this challenge, we count the number of requests that are not filtered due to the dispersion of the prefetch queues.

3.3 Infrastructure for dynamic profiling

To quantify this challenge, we included two new arrays of statistics for each prefetch engine in the simulator. This set of statistics counts, for each tile, the useful and non-useful prefetches issued by all the prefetchers to this tile and preserves the information of the tile that has issued the prefetch request. Note that when a cache line is evicted from the cache, its prefetching associated information is checked. If the line was a prefetched line, it will become an useful or unuseful prefetch. Then, according to the issuer of the operation the information is going to be aggregated in its corresponding accumulator. By aggregating the information from all the tiles, we can obtain dynamically accurate information about the behavior of each specific prefetch engine. To quantify the absolute error of the accuracy, we have compared two values in each

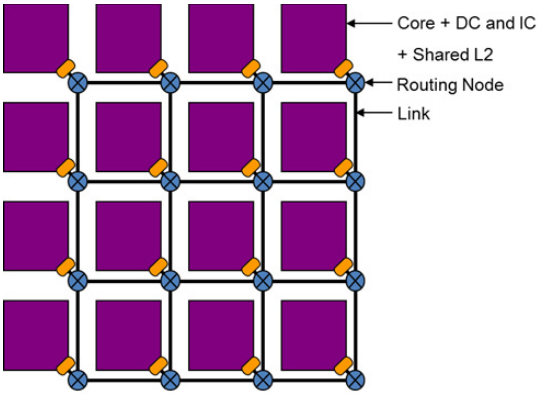
tile. The first one is the one that we have already commented, which is the realistic value of the accuracy in each tile. The second value is the result of aggregating all the values in the same tile. This would be the value obtained without doing any modification in the system. However, this second statistic accounts for requests from different prefetches and is not representative of any one in particular. For this reason this is not an accurate value.

4 Challenge analysis

In this section, we show the employed framework, the analysis of the challenges one by one, and quantification of the error that each can generate.

4.1 Experimental framework

The system simulated in this performance evaluation is represented in Figure 3a. Note that, in order to simplify the representation, there are only 16 of the 64 simulated cores. However, our simulations are done with 64 cores. The framework consists of a tiled mesh with private first level data and instruction caches, and a shared L2 cache. The L2 cache comprises several banks and each of these banks is associated with a local tile. Each memory address is deterministically associated with a given L2 location. This means that there is no replication in the L2 cache, though a given block can be in several L1 caches. L2 and L1 are not inclusive. The coherence protocol employed is the MOESI CMP directory. The prefetchers used to quantify the challenges are the well known tagged prefetcher [6] and the Global History Buffer (GHB) [8] prefetchers. According to the challenge we have used one prefetcher or the other one. In fact, to quantify the pattern detection challenge the GHB prefetcher has been chosen. Note that the tagged prefetcher does not use the stream of memory accesses and is therefore not affected by this challenge. However, to evaluate the queue filtering challenge or the dynamic profiling challenge we have used the tagged prefetcher because it is the one that is unaffected by the pattern detection challenge. The hardware specifications are shown in Table of Figure 3b. The simulator used for the analysis was gem5 [3] and the benchmark suite in this performance study was a subset of the PARSEC 2.1 [4].



(a) Tile configuration.

Hardware specifications	Values
ISA	x86
CPU model	TimingSimple
Number of tiles	64
L1 Data cache size	16KB per tile
L1 Instruction cache size	16KB per tile
L2 Unified cache size	16MB
Network	Garnet
Topology	Mesh
Prefetcher cache level	L2
Simulated cycles	350 millions
Tagged prefetch aggr/dist	2/2
GHB depth/width	8/8

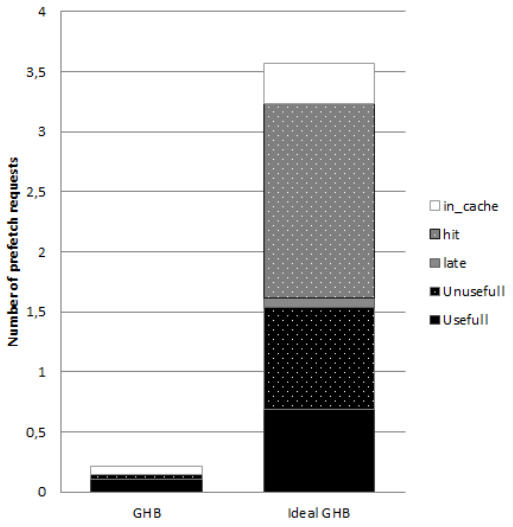
(b) Simulator specifications.

Figure 3: Simulation environment details.

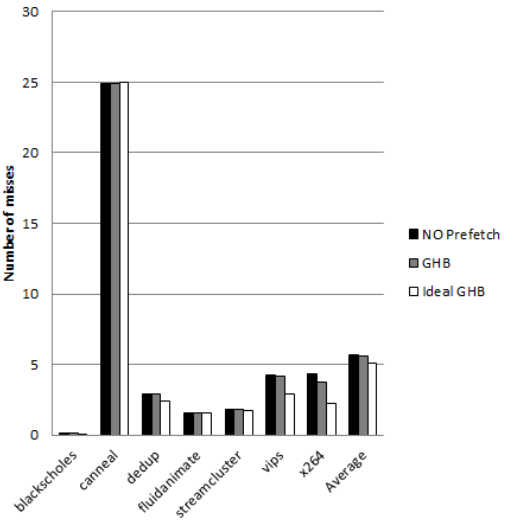
4.2 Pattern detection analysis

The GHB is a prefetcher that records the miss stream of the memory it is working with, analyzes it, and attempts to find a correlation between the last accesses. Figure 4a shows the evaluation of all the requests issued. The total value of the bars represents the number of generated requests every 1000 instructions. Note that a distributed GHB can only be aware of certain parts of the pattern. For this reason, it is unable to find correlation in the miss stream and is therefore unable to generate prefetch requests. The unified and ideal GHB therefore launches up to 6.5 times more useful prefetches than the distributed one.

Figure 4b shows the misses every 1000 instructions (MPKI). As a consequence of a prefetch engine that is not working properly, the MPKI is not reduced as much as it should be. We can see that in almost all the workloads the ideal GHB reduces the MPKI to a higher degree than the distributed GHB does.



(a) Evaluation of the generated requests by GHB and the ideal GHB for the pattern detection challenge analysis.



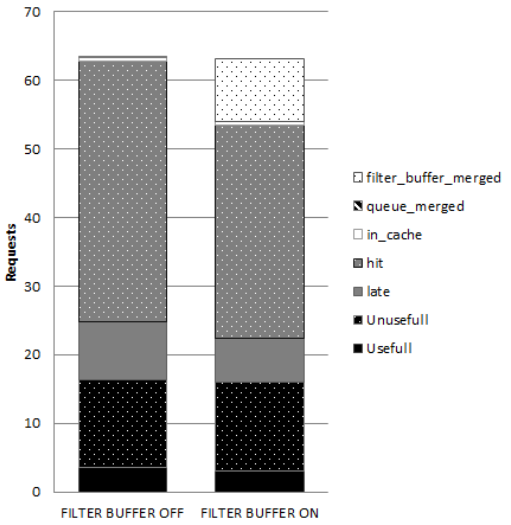
(b) MPKI in L2 without prefetching, GHB, and ideal GHB for the pattern detection challenge analysis.

Figure 4: Pattern detection challenge analysis.

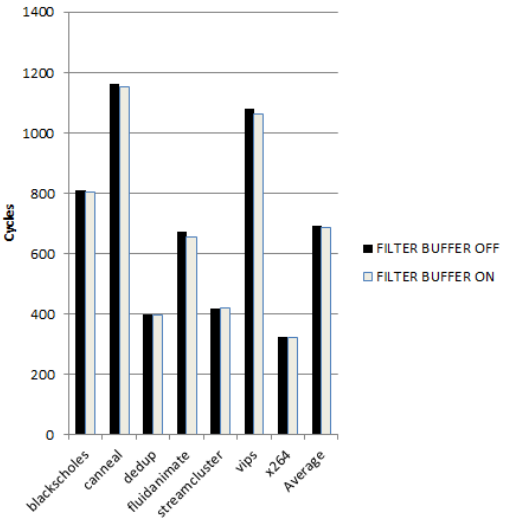
4.3 Prefetching queue filtering analysis

To analyze this challenge we used a filtering buffer. This buffer is aware of all the prefetch requests in all the prefetch queues. When queuing new requests, if the request is in any other queue of the system, it is merged. Figure 5a shows not only the issued requests but also the generated ones. The stacked bar with the filter buffer merge value represents the number of requests that the distributed prefetcher queue is not able to filter. For this reason, without the filtering buffer, the prefetcher will inject up to 30% more traffic into the network.

Figure 5b shows the average latency of a miss in L1. We can see that the filtering has a direct effect on performance. This is because the filtering effect reduces congestion in the network, which reduces the miss latency.



(a) Average number of generated requests by the Tagged prefetcher for the queue filtering.



(b) Average miss latency in L1 for all the benchmarks for the prefetching queue filtering.

Figure 5: Queue filtering challenge analysis.

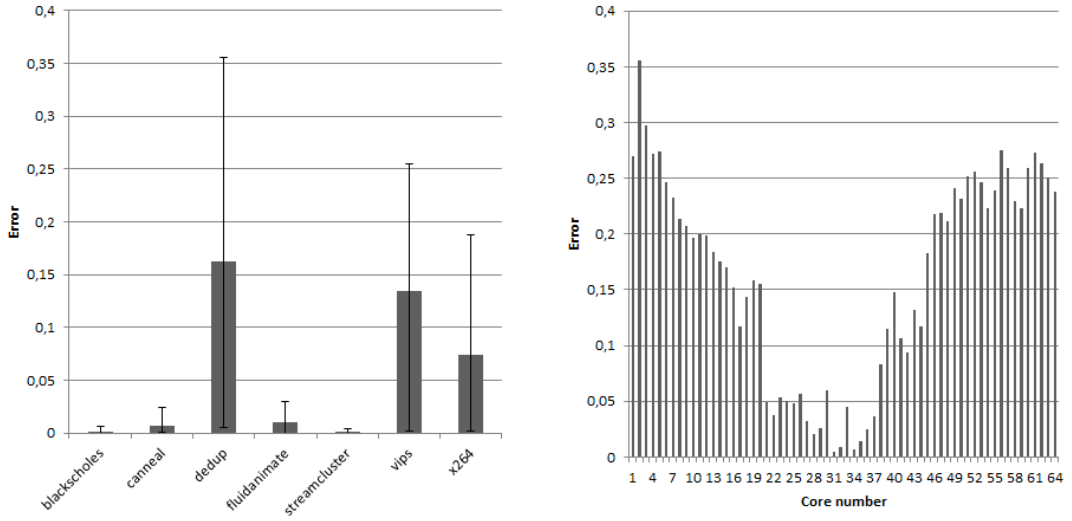
4.4 Dynamic profiling analysis

We calculated the absolute error between these two values and the result is shown in Figure 6a and Figure 6b. In Figure 6a, we can see two elements: the bars, whose value represents the average error among all the tiles and the segment bars, which represent the maximum and minimum errors among all the tiles in the same chip. We can see that the error for some benchmarks is relatively small. This happens because, depending on the behavior of the tagged prefetch, the requests issued from one tile are almost always addressed to the same tile and this effect balances out the statistic. However, in other benchamrks the error is more significant, specially when we analyze the error observed in each tile as we can see in Figure 6b.

To see the error in the various tiles in greater detail, Figure 6b shows the absolute errors for all the tiles in the dedup benchmark, which is the one with the greatest errors. We can see that the error can sometimes reach differences of over 35%.

5 Facing the challenges

Although it is not in the scope of this paper, we would like to hint the reader in how to face the challenges presented in this paper. There are mainly two directions to follow in order to face the challenges. The first one is to **adapt** the distributed architecture to emulate a unified memory system from the prefetcher point of view. The second option is to **redesign** the whole prefetching mechanism in order to find new smart techniques that are able to work in DSMs avoiding the challenges presented in this paper. Focusing on the pattern detection challenge, which is probably the most critical, a typical prefetching mechanism works by analyzing the data pattern of an application and predicting the next data blocks that are going to be require. As previously stated, in a DSM, a prefetching engine sees only a part of the data stream of different applications and potentially not in the same order as the operations were issued. A



(a) Absolute error in accuracy for the Tagged prefetcher for the dynamic profiling analysis.

(b) Absolute error for all the tiles with the dedup benchmark for the dynamic profiling analysis.

Figure 6: Dynamic profiling challenge analysis.

prefetching mechanism designed from scratch should be aware of these characteristics to be effective.

In any case, if the prefetcher is adapted or redesigned, there are two main design options that may be taken. Given that the challenges appear when the prefetching technique is applied to a DSM, a possible solution would be to **centralize** in somewhere the information related with the prefetcher (the memory accesses, the statistics, and the generated prefetch requests). If the prefetcher is able to work as a centralized module with a unified memory system, the challenges are not going to appear, and the prefetcher will work properly. Another option would be to design a smart **distributed** framework that is able to overcome the challenges by sharing the appropriate information among the prefetching engines assigned to each of the tiles.

Independently of the approach chosen, the designed framework will have to consider issues like the resulting size of the prefetcher data structures, the traffic that has to traverse the network on chip, the required processing throughput of the prefetching engines, the power consumption, etc. These issues in addition to the ability to prefetch accurately will decide the viability of the new technique.

6 Conclusions and future work

In this paper, we have shown the challenges when trying to prefetch in a distributed and shared memory system. We have described and quantified these challenges: (1) pattern detection, (2) prefetch queue filtering, and (3) dynamic profiling. We have also shown that the memory access pattern in the private cache, which is more or less regular and predictable, becomes totally irregular in the distributed memory. For this reason, the techniques used by the prefetcher to guess the future misses in the unified memories do not properly apply when these prefetch engines are distributed. Moreover, the dispersion of the prefetching queues reduces the potential

of the prefetchers. Finally, the techniques that use prefetch efficiency statistics from the caches to dynamically modify the behavior of the prefetching mechanism are also challenged. We believe that the experimental results provided in this work are important for the community, as we make strong evidence that when targeting this kind of architectures new approaches for prefetching are needed. Furthermore, we have proposed some hints on how these challenges can be solved. For this reason, as future work, this analysis opens the door to several works that wish to address these challenges and solve them or make feasible and efficient the techniques proposed on this study.

Acknowledgements

This work has been partially supported by the Spanish Ministry of Science and Innovation (MCI) and FEDER funds of the EU under the contracts TIN201018368 and TIN201347245C22R, and the Generalitat of Catalunya under grants 2009SGR1250 and 2013FIB100127.

References

[1] Levinthal D. Performance analysis guide for Intel Core i7 processor and Intel Xeon 5500 processors. *White paper (2009)*, 2009.

[2] Eiman Ebrahimi, Onur Mutlu, Chang Joo Lee, and Yale N Patt. Coordinated control of multiple prefetchers in multi-core systems. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 316–326. ACM, 2009.

[3] Binkert Nathan et al. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[4] Christian Bienia et al. The parsec benchmark suite: Characterization and architectural implications. 2008.

[5] Nachiappan Chidambaram Nachiappan et al. Application-aware prefetch prioritization in on-chip networks. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 441–442. ACM, 2012.

[6] John W. C. Fu, Janak H. Patel, and Bob L. Janssens. Stride directed prefetching in scalar processors. *SIGMICRO Newsl.*, 23(1-2):102–110, December 1992.

[7] Junghoon Lee, Hanjoon Kim, Minjeong Shin, John Kim, and Jaehyuk Huh. Mutually aware prefetcher and on-chip network designs for multi-cores. 2013.

[8] K.J. Nesbit and J.E. Smith. Data cache prefetching using a global history buffer. In *Software, IEEE Proceedings-*, pages 96–96, 2004.

[9] Byna S., Yong C., and Xian-He S. Taxonomy of data prefetching for multicore processors. *Journal of Computer Science and Technology*, 24:405–417, 2009.

[10] S. Srinath, O. Mutlu, Hyesoon Kim, and Y.N. Patt. Feedback directed prefetching: Improving the performance and bandwidth efficiency of hardware prefetchers. In *In IEEE 13th International Symposium High Performance Computer Architecture, 2007. (HPCA).*, pages 63–74, 2007.

[11] Tilera. Tile-gx processor family webpage. `http://www.tilera.com/products/processors/TILE-Gx_Family/`, 2014. [Online].

[12] Martí Torrents et al. Comparative study of prefetching mechanisms. *CEDI*, 2012.

[13] Xiaotong Zhuang and H-HS Lee. A hardware-based cache pollution filtering mechanism for aggressive prefetches. In *Parallel Processing, 2003. Proceedings. 2003 International Conference on*, pages 286–293. IEEE, 2003.

[14] Xiaotong Zhuang and Hsien-Hsin S. Lee. Reducing cache pollution via dynamic data prefetch filtering. *IEEE Trans. Comput.*, 56(1):18–31, January 2007.